

A Framework for Extending microKanren with Constraints

Jason Hemann

Daniel P. Friedman

Indiana University
Indiana, USA

{jhemann,dfried}@indiana.edu

We present a framework for building CLP languages with symbolic constraints based on microKanren, a domain-specific logic language shallowly embedded in Racket. We rely on Racket’s macro system to generate a constraint solver and other components of the microKanren embedding. The framework itself and the constraints’ implementations amounts to just over 100 lines of code. Our framework is both a teachable implementation for CLP as well as a test-bed and prototyping tool for symbolic constraint systems.

1 Introduction

Constraint logic programming (CLP) is a highly declarative programming paradigm applicable to a broad class of problems [19]. Jaffar and Lassez’s CLP scheme [17] generalizes the model of logic programming (LP) to include constraints beyond equations. Constraints are interpreted as statements regarding some problem domain (e.g., a term algebra) and a constraint solver evaluates their satisfiability. The CLP scheme explicitly separates constraint satisfiability from inference, control, and variable management.

While CLP languages’ semantics may respect this separation, their implementations often closely couple inference and constraint satisfaction to leverage domain-specific knowledge and improve performance. Such specialization and tight coupling could force an aspiring implementer to rewrite a large part of the system to integrate additional constraints. LP metainterpreters, another common approach to implementing CLP systems, are unsatisfying in different ways. The user pays some price for the interpretive overhead, even if we use techniques designed to mitigate the cost [26]. Moreover, implementing intricate constraint solvers in a logic language can be unpleasant, and performance concerns may pressure implementers to sacrifice purity. The complexity in existing Kanren implementations that support various symbolic constraints [6] is also costly in other ways.

miniKanren is a pure logic language implemented as a purely functional, shallow embedding in a host language, e.g. Racket [10]. microKanren [13] is an approach to clarifying miniKanren’s complexities. It separates the core implementation from the surface syntax, and is just over 50 lines of code in length. Such separation allows functional programmers in call-by-value languages to implement the core logic programming features without the syntactic sugar. There are small syntactic differences between the miniKanren and microKanren languages and their various implementations. We elide these details and describe them collectively as “Kanrens” unless otherwise relevant. In all cases a Kanren embedding gives functional programmers access to logic programming in languages lacking native support.

The original microKanren is equipped only with a syntactic equality constraint, but many of the most interesting typical uses of Kanrens require symbolic constraints [6] beyond equality, including disequality constraints, subterm constraints, and a variety of sort constraints. Implementations with these additions have ballooned to upwards of a thousand lines of code, and seem somewhat baroque compared to the 50-line microKanren. Some of this added heft comes from constraint solving; the remainder involves transforming constraint sets to solved form and answer projection. In such versions,

implementing each new kind of constraint in the host language requires plumbing domain knowledge throughout the implementation on an ad-hoc basis, and complicates constraint solving, subsumption, reduction to solved form, and answer projection. In short, it obscures what began as an imminently teachable artifact.

We provide a framework for building shallowly-embedded Racket implementations of microKanren-like CLP languages with symbolic constraints. The language designer provides the atomic constraints' intended interpretations (implemented as Racket predicates) as inputs to the framework. We then use Racket macros to generate a functional CLP embedding from the language designer's input. The generated implementations mirror the separation of inference and solving in the CLP scheme semantic model. Our macros generate a constraint solver, as well as other components of the embedding. Our framework's generated languages return solutions as bags of constraints. We intend to extend the framework to also reduce constraints to solved, canonical forms, eliminate redundancies, and project answers with respect to initial query variables. In the extension we envision, the language designer describes rewriting-rules for constraint sets, and the framework does the rest. We envision using our framework as a tool for rapidly prototyping constraints and CLP languages, in addition to being an educational artifact for functional programmers.

We describe the Kanren term language, the domain of our symbolic constraints, in Section 3. We use this framework to implement common miniKanren constraints in Section 6, as well as suggestive new ones in Section 7. Our untyped shallow embedding gives the language designer a fair amount of flexibility, and we describe future directions and alternate design choices in Section 9. This paper is a literate document; it contains the full implementation of our framework in Section 5, and we provide the inference engine in roughly 30 lines of code in the Appendix. The complete framework and the implementation of common miniKanren symbolic constraints comprise just over 100 lines—a marked decrease in line count over similarly featureful implementations. We also provide our full implementation at github.com/jasonhemann/constraint-microKanren alongside several syntax extensions implemented as macros on top of the core syntax. To recapitulate, our contributions include:

- A macro-based framework for generating pure, functional, shallowly-embedded CLP languages in Racket;
- The implementations of several symbolic constraints common to Kanren;
- The implementations of several more suggestive and useful symbolic constraints; and
- A literate presentation of the complete framework and the implementation of our constraints.

2 Background

We do not expect any miniKanren background or experience of the reader; a logic programming background is enough. We briefly adumbrate here the necessary background material. We review some features and contrast miniKanren's syntax and behavior to that of Prolog. Interested readers should consult the references, tutorials, and myriad implementations at miniKanren.org for further details.

Although Kumar [25] has formally specified miniKanren's syntax and given several semantics, implementers have been encumbered by neither, and none of the more widely used implementations obey these specifications [29, 4]. Instead, miniKanren is better described as a family of related logic programming languages, traditionally shallowly embedded in a declarative host language, most of whose semantics are informally specified by direct appeal to their host languages' features. Kanrens inherit much of the syntax and structure of their hosts. Implementations' concrete syntax varies from host to

host because of the shallow embedding. Different miniKanren implementations often provide different extensions and operators, as happens with Prologs. The traditional Prolog definition of a naïve reverse (`nrev`) is syntactically analogous to the miniKanren version defined using pattern-matching syntax [21].

```

nrev([], []).
nrev([H|T], L2) :- nrev(T, R),
                  append(R, [H], L2).

(defmatche (nrev l1 l2)
  (((() ()))
   (((,h . ,t) ,l2)
    (fresh (r)
     (nrev t r)
     (append r ‘(,h) l2))))))

```

Rather than using case to distinguish constants and variables, in our embedding symbols in quoted (‘) data are constants and are otherwise considered variables. In pattern matches, we precede variables by an unquote (,), and terms are otherwise considered constants. Pairs are destructured as (, α . , β) rather than $[\alpha|\beta]$. Predicates are defined at once as a collection of clauses, and we do not require the name of the predicate in every clause. Unlike in Prolog, the programmer must explicitly introduce auxiliary variables using the `fresh` operator.

Below is the translation of the `nrev` into microKanren. The operators `conj` and `disj` provide binary conjunction and disjunction; `call/fresh` introduces scope, relying on Racket’s λ for lexical binding. The operator `==` is microKanren’s first-order syntactic equality constraint. Finally, the operator `define-relation` defines user predicates and plays a part in the interleaving. Syntactically, the language resembles Spivey and Seres’s Haskell embedding of Prolog [31]. Modulo differences in syntax, both essentially use *completed predicates*, à la Clark [5]. We can layer the miniKanren syntactic sugar over this somewhat verbose core syntax with about 45 lines of Racket macros. This approach is now the most common way to implement miniKanren.

```

(define-relation (nrev l1 l2)
  (disj (conj (== l1 ‘())
              (== l2 ‘()))
        (call/fresh
         (λ (h)
          (call/fresh
           (λ (t)
            (conj (== ‘(,h . ,t) l1)
                  (call/fresh
                   (λ (r)
                    (conj (nrev t r)
                          (append r ‘(,h) l2)))))))))))

```

The `run` operator is the way to evaluate a miniKanren expression. It takes a number (here 1), a variable name (here `q`), and a query to execute. The `run` operator returns a list of at most that many answers to the query, simplified and answer projected [20] with respect to the query variable. Below we demonstrate `nrev` queries in Prolog and miniKanren.

```

?- findnsols(1,L2,nrev([a,b,c],L2),Q).      > (run 1 (q) (nrev ‘(a b c) q))
Q = [[c,b,a]] ?                             ‘((c b a))

```

Prologs default to depth-first search, whereas Kanrens rely on an unguided, interleaving depth-first search, based on Kiselyov et al.’s Logic monad [24], that is both complete and more useful in practice than are breadth-first or iterative deepening depth-first search. Other than Kanrens, this style of search

is not currently available in any logic language of which we are aware. This complete, more efficient search makes a purely relational approach to programming more practical, as we can query pure, all-modes relations more effectively. This in turn reduces the necessity of non-logical operators.

For instance, Kanrens have no equivalent to Prolog’s `is` operator. Instead, the Kanren arithmetic system is built from all-moded relations built up from bit-adders, without any use of non-logical operators [23]. The `add` used in the following query is a 3-place addition relation and `log` is a four-place logarithm relation. `miniKanren` prints the answers as little-endian binary numbers. In the first, we return two answers, and indeed $0 + 0 = 0$ and $1 + 1 = 2$. In the second, we return a list of the only answer `(#f #t #t)`, as $14 = 2^3 + 6$.

```
> (run 2 (a b) (add a a b))
'((( () ) ((#t) (#f #t))))
> (run* (q) (log (#f #t #t #t) (#f #t) (#t #t) q))
'((#f #t #t))
```

These relations run in all modes, and with our interleaving search appropriately terminate. This declarative arithmetic is efficient enough to be useful in practice [2]. Kanrens do not currently support arithmetic over real or floating-point numbers, and heavily numeric computations are not Kanrens’ strong suit. Instead, this all-moded logic programming technique inspires non-traditional sorts of symbolic computation. Such Kanren programming examples include a typechecker that also behaves as a type inhabitant [28], an automated theorem prover that doubles as a proof assistant [3], and a programming-language interpreter that also serves as a quine generator [4]. Many of these `miniKanren` examples are available in the browser at tca.github.io/veneer/examples/editor. Consider as a representative example `eval`, a relational interpreter for a Racket-like language. The predicate holds between an expression `e`, an environment ρ , and a value `v` when the value of `e` in ρ is `v`. To search for a quine, we query `eval` for an expression that evaluates to its listings (source code). The result is a valid Racket quine.

```
> (run 1 (q) (eval q '() q))
'((((λ (_ .0) (list _ .0 (list 'quote _ .0)))) (λ (_ .0) (list _ .0 (list 'quote _ .0))))
  (=/= ((_ .0 closure)))
  (sym _ .0)))
```

`miniKanren` prints fresh variables as `_ .n` in projected answers. The above answer is subject to several constraints: `_ .0` must be a symbol other than `closure`. Historically, `miniKanren` implementers add new kinds of constraints in response to challenges that arise in the course of problem solving.

3 Kanren terms algebra

The Kanren term language contains symbols, Booleans, the empty list `()`, logic variables, and cons pairs of the above. In the interest of simplicity we reserve non-negative integers as logic variables. Constraints are interpreted directly in the term structure. Our syntactic equality constraints, built with `==`, are equations in free F -algebra over our logic variables. The additional common Kanren atomic constraint operators are binary term disequality, written `=/=`, binary subterm discontainment, written `absento`, and the unary sort constraints `symbolo`, and `not-pairo`¹. These last two declare the constrained term a symbol or a non-pair respectively. We term “constraints” the finite conjunction of these atomic constraints.

¹Implementations often also include numeric constants and a sort constraint `numero`. We omit them here; their implementation follows naturally.

The meanings of these primitive relation symbols beyond `==` are given by the language designer as a collection of host-language predicates. We first decide the satisfiability of the conjunction of the `==` constraints, either failing or producing a substitution. Relying on the independence of negative constraints, we then use the provided predicates to determine the satisfiability of the remaining constraints modulo that substitution. We check all atomic constraints of the same kind at once, checking each kind in turn.

In addition to their usual benefits, our constraints also allow us to compress what would be multiple answers (potentially infinitely many) into single finite representations. Consider for instance, the `absento` constraint. An `absento` constraint holds between two terms x and y when x is neither equal to, nor a subterm of y . With just `=/=` constraints, we can in general only express this relationship in the limit, e.g. an infinite conjunction of disequalities between the fresh variable y and all possible terms x from which it is absent. With the `absento` constraint we can represent this relationship finitely.

4 Constraint framework user's requirements

A CLP-language designer is the *user* of our framework. They provide as inputs to the framework's macros their language's atomic constraint relation symbols, as Racket symbols, and the conditions that violate constraints, via predicates to test for invalid sets of constraints. From these, the framework will generate the microKanren (and thus miniKanren) constraint operators and a constraint solver automatically. They are a declarative, functional specification of what it means to violate these constraints. Constraint-violation predicates, *qua* predicates, are by definition total functions. As such, the solver for the constraint system (`invalid?`, defined in Section 5) is also total.

We provide `==`, representing syntactic first-order equality, with every constraint system, and implement it via unification with `occurs?` check (see Appendix). We fix this particular equational theory because of our intended use cases for the generated languages (e.g. sound type inference in a simply-typed language).

We require the resultant constraint solver to be *well-behaved* [18]. This means it is *logical*—that is, it gives the same answer for any representation of the same constraint information (i.e., regardless of order, redundancy, etc). It is also *monotonic*—that is, for any set of constraints, if the solver deems the set invalid, adding additional constraints cannot produce a valid set. Therefore, when adding a new constraint-violation predicate, a language designer need not modify older ones. Such a redesign may, however, clarify these violations. Presently, these requirements are unchecked.

5 Constraint framework implementation

In this section we completely implement our framework. We implement the constraint store as a persistent hash table in the host language. Each type of atomic constraint operator in the embedded language is a distinct key/value pair in the hash table. We use the relation symbol as that relation's key in the store. We define the initial state (constraint store) S_0 as an immutable hash table with `==` and each of the provided constraint identifiers as keys associated with `()`, Racket empty lists.

```
(define S0 (make-immutable-hasheqv '==(cid) ...))
```

Since symbols for distinct relations are unique, each field will have a distinct key. The hash table is immutable to allow structure sharing across different extensions of the same store, and we rely on the host language's garbage collection to free memory.

```
> (define == (make-constraint-goal-constructor '==))
...
```

Invoking `make-constraint-goal-constructor` yields the implementation of each relation in our embedding. We use the relation's name as its key in the store. `make-constraint-goal-constructor` takes a field in the store and returns a function accepting the correct number of term arguments. Such functions are the definitions of relations in our embedding.

```
(define ((make-constraint-goal-constructor key) . ts) S/c)
  (let ((S (ext-S (car S/c) key ts)))
    (if (invalid? S) '() (list '(', S . , (cdr S/c))))))
```

Invoking this relation with terms yields a goal: a function expecting a store and returning a stream of stores. To evaluate a constraint we extend the store by adding the newly constrained terms and test the store's consistency. If the extended constraint store is consistent, we return a stream of a single store; if not, we return the empty stream. Once added, constraints are not removed from the store. This decision means the size of the constraint store and the cost of checking constraints grows each time we encounter a constraint in the execution of a program. In Section 9 we suggest improvements.

The `ext-S` function takes the store, the key, and a list of terms. The `ext-S` function adds those terms, as a data structure, to a list of such structures. By consing all of the terms together, `hash-update` creates the data structure.

```
(define (ext-S S key ts) (hash-update S key ((curry cons) (apply list* ts))))
```

We check consistency with `invalid?`. `make-invalid?` builds the definition of `invalid?`. The language designer provides `make-invalid?` a list of the relation symbols (Racket identifiers). The designer also provides a sequence of predicates that check for constraint violations. Each predicate takes a substitution and returns true if it detects a violation. The constraint identifiers are free variables of the predicates; the expansion of `make-invalid?` will bind them. The result of `make-invalid?` is a predicate that tests if a store is invalid.

The Racket primitive `define-syntax-rule` builds a macro. This macro transforms an occurrence of the pattern, an expression beginning with `make-initial-state` followed by zero or more identifiers into an instantiation of the macro's template.

```
(define-syntax-rule (make-invalid? (cid ...) p ...)
  (λ (S) (let ((cid (hash-ref S 'cid))) ...)
    (cond ((valid== (hash-ref S '==)) => (λ (s) (or (p s) ...)))
          (else #t)))))
```

The first relation we check is `==`. If these constraints are consistent, the result is a substitution that is the m.g.u. of these constraints. Assuming this field is valid, we pass the resulting substitution as an argument to the constraint-violation predicates.

Our framework includes the implementation of the relation `==` and provides `==` in every generated constraint system. The `==` relation is special because we consider the satisfiability of other relations modulo this equivalence. For the remaining relations, we can apply the substitution to the constrained terms and check for satisfiability in the free term algebra generated by the set of variables away from the domain of the substitution. The `valid==` function below and its associated help functions are also included with the framework. The `valid==` function expects a list of cons pairs of terms to unify with each other. We define `unify` in the Appendix.

```
(define (valid== ==)
  (foldr (λ (pr s) (and s (unify (car pr) (cdr pr) s))) '() ==))
```

This is the main syntactic form for building constraint systems. We build the entire constraint system and embedded language with one invocation of `make-constraint-system`. This new syntactic form takes the same parameters as does `make-invalid?`. It builds `invalid?`, the initial store, and all the functions implementing the relations themselves. The result is a constraint system; together with `microKanren`'s control infrastructure (see Appendix) this yields a full implementation of a `microKanren`-like CLP language. To construct a `microKanren` with just equality, the language designer invokes `make-constraint-system` with an empty list of relation identifiers and no violation predicates.

```
> (make-constraint-system ())
```

The definition below uses Racket's `syntax-parse` [9], a more sophisticated macro system. We pattern-match on the `syntax` argument, and the hash (`#`) begins the definition of the syntax template. We use `syntax-local-introduce` to introduce three new identifiers into lexical scope; the remaining constraint identifiers are already scoped.

```
(define-syntax (make-constraint-system stx)
  (syntax-parse stx
    [(_ (cid:id ...) p ...)
     (with-syntax ([invalid? (syntax-local-introduce #'invalid?)]
                   [S0 (syntax-local-introduce #'S0)]
                   [== (syntax-local-introduce #'==)])
       #'(begin (define invalid? (make-invalid? (cid ...) p ...))
                 (define S0 (make-immutable-hasheqv '==(cid) ...))
                 (define == (make-constraint-goal-constructor '==))
                 (define cid (make-constraint-goal-constructor 'cid))
                 ...)))]))
```

This macro is the primary driver of our framework. The preceding code and the half page of code in the Appendix comprise the entire implementation.

6 Implementing a constraint system

Next, we make further use of our framework. We implement a series of violation predicates with some associated help functions and use those predicates to generate a constraint system for common symbolic constraints. We develop these relations and their predicates one at a time.

The typical `Kanren` contains four other relations beyond `==`. These are `=/=`, `absento`, `symbolo`, and `not-pairo`. We discuss the predicates required to implement these relations one at a time.

We first add a predicate to test for a violated `=/=` constraint. This predicate searches for an instance where, with respect to the current substitution, two terms under a `=/=` constraint are already equal. In that case, the `=/=` constraint is deemed violated.

```
> (make-constraint-system (=/= absento symbolo not-pairo)
  (λ (s) (ormap (λ (pr) (same-s? (car pr) (cdr pr) s)) =/=:))
  ...)
```

We implement this predicate in terms of a help function `same-s?`. If the result of unifying two terms in the substitution is the same as the original substitution, then those terms were already equal relative to that substitution.

```
#| Term × Term × Subst → Bool |#
(define (same-s? u v s) (equal? (unify u v s) s))
```

The next predicate checks for violated `absento` constraints, using the auxiliary predicate `mem?`. The predicate searches for an instance where, with respect to the substitution, the first term of a pair is already equal to (a subterm of) the second term. In that case, we deem the `absento` constraint violated.

```
> (make-constraint-system (=/= absento symbolo not-pairo)
  ...
  (λ (s) (ormap (λ (pr) (mem? (car pr) (cdr pr) s)) absento))
  ...)
```

The predicate `mem?` checks if a term `u` is already equivalent to any subterm of a term `v` under a substitution `s`. It makes use of `same-s?` in the check. If the result of unifying `u` and `v` is the same as the substitution `s` itself, then the two terms are already equal.

```
#| Term × Term × Subst → Bool |#
(define (mem? u v s)
  (let ((v (walk v s)))
    (or (same-s? u v s) (and (pair? v) (or (mem? u (car v) s) (mem? u (cdr v) s))))))
```

We write a third violation predicate to search for a violated `symbolo` constraint. For each term under a `symbolo` constraint, we look if that term, relative to the substitution, is anything but a symbol or a variable. If so, that term violates the constraint. We define `walk`, a function that performs a deep lookup of a term in a substitution, in the Appendix.

```
> (make-constraint-system (=/= absento symbolo)
  ...
  (λ (s) (ormap (λ (y)
    (let ((t (walk y s)))
      (not (or (symbol? t) (var? t)))))
    symbolo))
  ...)
```

The `not-pairo` violation predicate operates similarly. This last definition completes our implementation of the symbolic constraints common to Kanrens.

```
> (make-constraint-system (=/= absento symbolo not-pairo)
  ...
  (λ (s) (ormap (λ (n)
    (let ((t (walk n s)))
      (not (or (not (pair? t)) (var? t)))))
    not-pairo)))
```

We show below the execution of an example microKanren program that uses all the typical kinds of Kanren constraints. The result of invoking this program is a stream containing a single store. We see that all the constraints are present in the constraint store, and we can read off each constraint. The `#hasheqv(...)` is the printed representation of the hash table, whose elements are the key/value pairs. For instance, the `=/=` field, `(=/= . ((c . 0) (0 . b)))`, contains the pairs `(c . 0)` and `(0 . b)`. These are the `=/=` constraints that have been added.


```

> (call/initial-state 1
  (call/fresh
    (λ (x)
      (conj (== 'a x)
        (conj (/= x 'b)
          (conj (absento 'b '(,x))
            (conj (not-pairo x)
              (conj (symbolo x)
                (/= 'c x))))))))))
'((#hasheqv((= . ((a . 0))) (/= . ((c . 0) (0 . b))) (absento . ((b 0)))
  (symbolo . (0)) (not-pairo . (0)))
  . 1))

```

7 Adding new constraints

Beyond refactoring existing implementations, our framework also simplifies describing more complicated symbolic constraints new to Kanren: `booleano` and `listo`. The first mandates that the constrained term be a Boolean, and the second a proper list. These relations have more complex interactions than do the previous ones, and we need several new predicates to support each of these relations' implementation.

We add support for these constraints both because of their additional complexity and also their utility. With them, we can improve the implementations of relational interpreters, one of the archetypal miniKanren programming examples. Consider the partially-completed miniKanren definition of the relational interpreter `eval` below.

```

(defmatche (eval e ρ v)
  ((,e ,ρ ,v) (fresh () (symbolo e) (lookup e ρ v)))
  ((,e ,ρ ,v) (fresh () (booleano e) (listo ρ)))
  ...)

```

If `e` is a variable, `v` is its value in the environment `ρ`. We define `lookup` recursively as a three-place user predicate. When the variable is found in the environment, we return its value. In prior implementations of relational interpreters, the remainder of the environment remains unconstrained. Without `listo` constraints, the only way to ensure environments are proper lists requires generating via yet another user predicate the proper lists of all given lengths. Instead, we can now express infinitely many answers with a single `listo` constraint. We have more tightly constrained the implementation of `lookup`, which results in more precise answers.

```

(defmatche (lookup x ρ o)
  ((,x ((,x . ,o) . ,d) o) (listo d))
  ((,x ((,aa . ,da) . ,d) o) (fresh () (/= aa x) (lookup x d o)))

```

In prior definitions of `eval` [4], rather than using a `booleano` constraint, we equated the term first with `#t`, and then separately with `#f`. This generates near-duplicate programs that differ in their placement of `#t` and `#f`. By instead “compressing” the Booleans into one, we ensure the programs we generate have a more interesting variety.

7.1 Implementing `booleano`

Checking `booleano` involves more work than does checking our earlier sort constraints, since Boolean values are sort limited. The first predicate holds if we have forbid a term from being either of the

constants #t and #f while demanding that it be a Boolean. We also need a predicate to check for a booleano-constrained term that is a non-variable, non-Boolean. Finally since the booleano domain constraint is incompatible with symbolo, the last predicate checks for terms constrained by both.

```
> (make-constraint-system (=/= absento symbolo not-pairo booleano)
  ...
  (let ((not-b (λ (s) (or (ormap (λ (pr) (same-s? (car pr) (cdr pr) s)) =/=)
                              (ormap (λ (pr) (mem? (car pr) (cdr pr) s)) absento))))))
    (λ (s) (ormap (λ (b) (let ((s1 (unify b #t s)) (s2 (unify b #f s)))
                          (and s1 s2 (not-b s1) (not-b s2))))
                  booleano)))
  (λ (s) (ormap (λ (b) (let ((b (walk b s)))
                          (not (or (var? b) (boolean? b)))))
                  booleano)))
  (λ (s) (ormap (λ (b) (ormap (λ (y) (same-s? y b s)) symbolo))
                  booleano)))
```

The following is an example of its use.

```
> (call/initial-state 1
  (call/fresh
    (λ (x)
      (conj (=/= #f x)
            (conj (=/= #t x)
                  (booleano x))))))
'()
```

7.2 Implementing listo

Checking listo is more complicated still. Consequently some of the violation predicates are also quite complex. We add four independent predicates to properly implement listo. We briefly describe their behavior, and then provide their implementations.

In the first of these functions, we look for an instance in which the end of a term labeled a proper list l is required to be a symbol. We use the help function walk-to-end in constraint-violation predicates related to listo constraints. This help function recursively walks the cdr of a term x in a substitution s and returns the final cdr of x relative to s.

```
#| Term × Subst → Bool |#
(define (walk-to-end x s)
  (let ((x (walk x s)))
    (if (pair? x) (walk-to-end (cdr x) s) x)))
```

The second predicate resembles the first, except it checks for a Boolean instead. In the third, we check for a proper list that must have a definite fixed last cdr (the end) under the substitution. This means either end already is (), or a not-pairo constrains end. If, in addition, either =/= or absento constraints forbid end from being (), then that is a violation.

In the last predicate we require in order to correctly implement listo, end can be a proper list of unknown length. An absento constraint forbidding () from occurring in a term containing end, however, causes a violation. The constraint must precisely forbid () from occurring in a term containing end to cause the violation.

```

> (make-constraint-system (=/= absento symbolo not-pairo booleano listo)
  ...
  (λ (s) (ormap (λ (l) (let ((end (walk-to-end l s)))
                        (ormap (λ (y) (same-s? y end s)) symbolo)))
    listo))
  (λ (s) (ormap (λ (l) (let ((end (walk-to-end l s)))
                        (ormap (λ (b) (same-s? b end s)) booleano)))
    listo))
  (λ (s) (ormap (λ (l) (let ((end (walk-to-end l s)))
                        (let ((ŝ (unify end '() s)))
                          (and ŝ
                            (ormap (λ (n) (same-s? end n s)) not-pairo)
                            (or (ormap (λ (pr) (same-s? (car pr) (cdr pr) ŝ))
                                /=)
                                (ormap (λ (pr) (mem? (car pr) (cdr pr) ŝ))
                                  absento)))))))
    listo))
  (λ (s) (ormap (λ (l) (let ((end (walk-to-end l s)))
                        (ormap (λ (pr) (and (null? (walk (car pr) s))
                                           (mem? end (cdr pr) s)))
                          absento)))
    listo))
  ...))

```

These violation predicates are somewhat involved—of necessity. We have ensured that constraint violations can each be treated independently and that they comprise the entirety of the constraint domain knowledge required. Furthermore, by requiring that our solver be monotonic and logical, we have ensured that adding new constraints never requires the language designer to modify existing predicates. Below is an example of a sample constraint microKanren programs using these new forms of constraints. We can still provide miniKanren syntax to the language user with with but a handful of host-language macros [13].

```

> (call/initial-state 1
  (call/fresh
    (λ (x)
      (conj (listo x)
        (conj (not-pairo x)
          (disj (=/= '() x)
            (absento x '()))))))))
'()

```

8 Related work

The modern development of CLP languages begins in the mid 1980s by groups in Melbourne, Marseilles, and the ECRC. The CLP scheme [17] is an important development from this era. The CLP scheme separates the inference mechanism from constraint handling and satisfaction. It subsumes many individual logic programming extensions and provides a theoretical foundation for disparate CLP languages.

CLP over the domain of finite trees has been thoroughly investigated. Maher [27] presents fundamental LP results translated to the context of CLP, and specifically formulates many standard LP results

in the context of CLP over finite trees. See also Comon et al. [8] for a survey of constraint solving on terms, including finite and infinite trees.

Unlike constraint-handling rules (CHR) based approaches [12], our framework does not rewrite or transform the constraint set when solving. Our approach does not utilize rewrite rules; we take a *semantic*, rather than syntactic, approach [7]. Our solvers interpret constraints directly in the term model to discover violations.

Schrijvers et al. offer a different motivation for separating constraint solving and search [30]. They implement different advanced search strategies via monad transformers over basic search monads. It's not yet clear where miniKanren's interleaving DFS search fits into their framework, although this is a topic we are currently investigating.

microKanren (and thus also miniKanren) is closely connected to pure Prolog. Spivey and Seres [31] embed a similar subset of Prolog work on a Haskell embedding of Prolog. Kiselyov's "Taste of Logic Programming" [22], and of course Ralf Hinze's extensive work on implementations of Prolog-style backtracking [14, 15] are all closely related.

There exists a different sort of CLP paradigm based on research in constraint satisfaction problems using constraint propagation to reduce the search space. cKanren, an earlier miniKanren for CLP, takes this different approach and uses domain restriction and constraint propagation [1]. Alvis et al. take as their primary example finite domains. Unlike languages generated by our framework, they minimize answer constraint sets and prettily format the results.

9 Conclusion

We have presented a framework for developing microKanren-like CLP languages in an instance of the CLP scheme. Decoupling the constraint management from the inference, control, and variable management has helped to clarify the behavior of microKanren. We support customary miniKanren constraints as well as interesting and useful new ones.

In our implementation we deliberately reject certain common optimizations and features that would have complicated our implementation. The solvers we generate do not simplify the constraint set as a consequence of solving. Indeed, these generated constraint solvers are not at all specialized for incremental constraint solving, to the point of even adding duplicate, wholly redundant, constraints. We do not minimize even the answer constraint set, nor project answers.

We do not intend to generate efficient, state-of-the-art CLP languages, and on that front we have surely succeeded. Instead of efficiency, our aim is a simple, general framework for implementing constraints in microKanren. We envision our framework as a lightweight tool for rapidly prototyping constraint sets. Language designers can explore and test constraint definitions and interactions without building or modifying a complicated and efficient dedicated solver. We also imagine it as an educational artifact that provides functional programmers a minimal executable instance of the CLP scheme.

Although we have preferred simplicity over performance here, we hope to investigate the performance impacts of various simple optimizations including incremental constraint solving, early projection [11], attributed variables [16], or calling out to an appropriate dedicated constraint solver. We hope to develop these optimizations as a series of correctness-preserving transformations.

In future work we also hope to build an extensible, generic constraint simplification framework analogous to our framework for building constraint solvers. The language designer should have to write only the individual constraint simplification rules for the framework to produce a simplifier. Ideally this framework will infer an efficient order in which to execute these simplification functions based on ab-

stract interpretation. We envision actually using a CHR approach in these simplifying solvers. We also want to formalize the meaning of a “kind” of constraint-violation. Defining precisely what violations a single violation predicate should check will clarify the language designer’s precise responsibilities.

As it exists, ours is a clear, simple framework for generating miniKanren languages with constraints and serves as a test-bed for developing constraint systems and an artifact of study. Further, it serves as a foundation for continued future work in designing constraint systems.

Acknowledgments

We thank Will Byrd, Chung-chieh Shan, and Oleg Kiselyov for early discussions of constraints in miniKanren. We thank Ryan Culpepper for his improvements to the framework macros. We also thank our anonymous reviewers for their suggestions and improvements.

References

- [1] Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd & Daniel P. Friedman (2011): *cKanren: miniKanren with Constraints*. *Scheme Workshop '11*.
- [2] Bernd Braßel, Sebastian Fischer & Frank Huch (2008): *Declaring Numbers*. *Electronic Notes in Theoretical Computer Science* 216, pp. 111–124. Available at <http://dx.doi.org/10.1016/j.entcs.2008.06.037>.
- [3] William E. Byrd & Daniel P. Friedman (2007): *α Kanren: A Fresh Name in Nominal Logic Programming*. In: *Proceedings of Scheme Workshop '07*, Université Laval Technical Report DIUL-RT-0701, pp. 79–90 (see also <http://webyrd.net/alphamk/alphamk.pdf> for improvements).
- [4] William E. Byrd, Eric Holk & Daniel P. Friedman (2012): *miniKanren, live and untagged*. In: *Proceedings of Scheme Workshop '12*, ACM. Available at <http://dx.doi.org/10.1145/2661103.2661105>.
- [5] Keith L. Clark (1978): *Negation as Failure*. In: *Logic and Data Bases*, Springer Science LNCS, pp. 293–322. Available at http://dx.doi.org/10.1007/978-1-4684-3384-5_11.
- [6] Hubert Comon (1994): *Constraints in Term Algebras (Short Survey)*. In: *Algebraic Methodology and Software Technology (AMAST'93)*, Springer Science LNCS, pp. 97–108. Available at http://dx.doi.org/10.1007/978-1-4471-3227-1_9.
- [7] Hubert Comon, Mehmet Dincbas, Jean-Pierre Jouannaud & Claude Kirchner (1999): *A Methodological View of Constraint Solving*. *Constraints* 4(4), pp. 337–361. Available at <http://dx.doi.org/10.1023/A:1009868906501>.
- [8] Hubert Comon & Claude Kirchner (2001): *Constraint Solving on Terms*. In: *Constraints in Computational Logics*, Springer Science LNCS, pp. 47–103. Available at http://dx.doi.org/10.1007/3-540-45406-3_2.
- [9] Ryan Culpepper (2012): *Fortifying macros*. *Journal of Functional Programming* 22(4-5), pp. 439–476. Available at <http://dx.doi.org/10.1017/s0956796812000275>.
- [10] Matthew Flatt & PLT (2010): *Reference: Racket*. Technical Report PLT-TR-2010-1, PLT Design Inc. <http://racket-lang.org/tr1/>.
- [11] Andreas Fordan (1999): *Projection in Constraint Logic Programming*. Ios Press.
- [12] Thom Frühwirth (2009): *Constraint Handling Rules*. Cambridge University Press (CUP). Available at <http://dx.doi.org/10.1017/cbo9780511609886>.
- [13] Jason Hemann, Daniel P. Friedman, William E. Byrd & Matthew Might (2016): *A small embedding of logic programming with a simple complete search*. In: *Proceedings of DLS '16*, ACM. Available at <http://dx.doi.org/10.1145/2989225.2989230>.

- [14] Ralf Hinze (2000): *Deriving backtracking monad transformers*. In: *Proceedings of ICFP '00*, ACM. Available at <http://dx.doi.org/10.1145/351240.351258>.
- [15] Ralf Hinze (2001): *Prolog's control constructs in a functional setting Axioms and implementation*. *International Journal of Foundations of Computer Science* 12(02), pp. 125–170, doi:10.1142/S0129054101000436. Available at <https://dx.doi.org/10.1142/S0129054101000436>.
- [16] Serge Le Huitouze (1990): *A new data structure for implementing extensions to Prolog*. In: *Programming Language Implementation and Logic Programming*, Springer Science LNCS, pp. 136–150. Available at <http://dx.doi.org/10.1007/bfb0024181>.
- [17] J. Jaffar & J.-L. Lassez (1987): *Constraint logic programming*. In: *Proceedings of POPL '87*, ACM. Available at <http://dx.doi.org/10.1145/41625.41635>.
- [18] Joxan Jaffar, Michael Maher, Kim Marriott & Peter Stuckey (1998): *The semantics of constraint logic programs*. *The Journal of Logic Programming* 37(1-3), pp. 1–46. Available at [http://dx.doi.org/10.1016/s0743-1066\(98\)10002-x](http://dx.doi.org/10.1016/s0743-1066(98)10002-x).
- [19] Joxan Jaffar & Michael J. Maher (1994): *Constraint logic programming: a survey*. *The Journal of Logic Programming* 19-20, pp. 503–581. Available at [http://dx.doi.org/10.1016/0743-1066\(94\)90033-7](http://dx.doi.org/10.1016/0743-1066(94)90033-7).
- [20] Joxan Jaffar, Michael J. Maher, Peter J. Stuckey & Roland H. C. Yap (1993): *Projecting CLP(\mathcal{R}) constraints*. *New Gener Comput* 11(3-4), pp. 449–469. Available at <http://dx.doi.org/10.1007/bf03037187>.
- [21] Andrew W. Keep, Michael D. Adams, Lindsey Kuper, William E. Byrd & Daniel P. Friedman (2009): *A Pattern-matcher for miniKanren -or- How to Get into Trouble with CPS Macros*. In: *Proceedings of Scheme Workshop '09*, Cal Poly Technical Report CPSLO-CSC-09-03, pp. 37–45.
- [22] Oleg Kiselyov (2006): *The taste of logic programming*. Available at <http://okmij.org/ftp/Scheme/misc.html#sokuza-kanren>.
- [23] Oleg Kiselyov, William E. Byrd, Daniel P. Friedman & Chung-chieh Shan: *Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl)*. In: *Functional and Logic Programming*, Springer Science LNCS, pp. 64–80. Available at http://dx.doi.org/10.1007/978-3-540-78969-7_7.
- [24] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman & Amr Sabry (2005): *Backtracking, interleaving, and terminating monad transformers: (functional pearl)*. In: *Proceedings of ICFP '05*, 40, ACM, pp. 192–203. Available at <http://doi.acm.org/10.1145/1086365.1086390>.
- [25] Ramana Kumar (2010): *Mechanising Aspects of miniKanren in HOL*. Australian National University. Bachelors thesis.
- [26] J. W. Lloyd & J. C. Shepherdson (1991): *Partial evaluation in logic programming*. *The Journal of Logic Programming* 11(3-4), pp. 217–242. Available at [http://dx.doi.org/10.1016/0743-1066\(91\)90027-m](http://dx.doi.org/10.1016/0743-1066(91)90027-m).
- [27] Michael J. Maher (1993): *A Logic Programming View of CLP*. In: *Proceedings of ICLP '93*, MIT Press, pp. 737–753.
- [28] Joseph P. Near, William E. Byrd & Daniel P. Friedman (2008): *α leanTAP: A Declarative Theorem Prover for First-Order Classical Logic*. In: *Proceedings of ICLP '08*, pp. 238–252. Available at http://dx.doi.org/10.1007/978-3-540-89982-2_26.
- [29] David Nolen (2016): *core.logic*. <https://github.com/clojure/core.logic>.
- [30] Tom Schrijvers, Peter Stuckey & Philip Wadler (2009): *Monadic constraint programming*. *Journal of Functional Programming* 19(06), p. 663. Available at <http://dx.doi.org/10.1017/s0956796809990086>.
- [31] J. M. Spivey & Silvija Seres (1999): *Embedding Prolog in Haskell*. In E. Meier, editor: *Proceedings of Haskell Workshop '99*, Utrecht University Technical Report UU-CS-1999-28. Available at <http://www.cs.uu.nl/research/techreps/repo/CS-1999/1999-28.pdf>.

Appendix: unification and microKanren control

Below are the implementation of unify and microKanren's control and variable management infrastructure. The first five lines implement variables and variable management. The next twenty lines implement substitutions and unify. The following six lines are the macro that implements define-relation, which plays a part in the interleaving, and the operators for conjunction and disjunction. The next twelve lines implement help functions for conj and disj that interleave streams. The subsequent twelve lines define ifte and once, impure microKanren operators that provide soft-cut and committed choice, respectively. The final ten lines actually run the computation, forcing a stream to mature, and also enable us to get a prefix from the mature, computed stream.

```
(define (var n) n)

(define (var? n) (number? n))

(define ((call/fresh f) S/c)
  (let ((S (car S/c)) (c (cdr S/c)))
    ((f (var c)) '(', S . , (+ 1 c)))))

(define (occurs? x v s)
  (let ((v (walk v s)))
    (cond ((var? v) (eqv? x v))
          ((pair? v) (or (occurs? x (car v) s)
                        (occurs? x (cdr v) s)))
          (else #f))))

(define (ext-s x v s)
  (if (occurs? x v s) #f '((,x . ,v) . ,s)))

(define (walk u s)
  (let ((pr (assv u s)))
    (if pr (walk (cdr pr) s) u)))

(define (unify u v s)
  (let ((u (walk u s)) (v (walk v s)))
    (cond ((eqv? u v) s)
          ((var? u) (ext-s u v s))
          ((var? v) (ext-s v u s))
          ((and (pair? u) (pair? v))
           (let ((s (unify (car u) (car v) s)))
             (and s (unify (cdr u) (cdr v) s))))
          (else #f))))

(define-syntax-rule (define-relation (r . as) g)
  (define ((r . as) S/c) (delay/name (g S/c))))

(define ((disj g1 g2) S/c)
  ($append (g1 S/c) (g2 S/c)))

(define ((conj g1 g2) S/c)
  ($append-map g2 (g1 S/c)))

(define ($append $1 $2)
  (cond ((null? $1) $2)
        ((promise? $1)
         (delay/name ($append $2 (force $1))))
        (else (cons (car $1)
                     ($append (cdr $1) $2)))))

(define ($append-map g $)
  (cond ((null? $) '())
        ((promise? $)
         (delay/name ($append-map g (force $))))
        (else ($append (g (car $))
                        ($append-map g (cdr $))))))

(define ((ifte g1 g2 g3) s/c)
  (let loop ((g1 s/c))
    (cond
      ((null? $) (g3 s/c))
      ((promise? $) (delay/name (loop (force $))))
      (else ($append-map $ g2)))))

(define ((once g) s/c)
  (let loop ((g s/c))
    (cond
      ((null? $) '())
      ((promise? $) (delay/name (loop (force $))))
      (else (list (car $)))))

(define (pull $)
  (if (promise? $) (pull (force $)) $))

(define (take n $)
  (cond ((null? $) '())
        ((and n (zero? (- n 1))) (list (car $)))
        (else (cons (car $)
                     (take (and n (- n 1))
                          (pull (cdr $)))))))

(define (call/initial-state n g)
  (take n (pull (g '(', S0 . 0)))))
```